Project #5
Computer Science 2334
Fall 2007

User Request: *"Develop a historical Hurricane Database."*

Milestones:

1. Use text file I/O to read and write text files. *(10 points)*
2. Create custom ADTs that are used to abstract the data, which are stored in lists; and create custom ADTs that abstract the usage of the lists. *(15 points)*
3. Implement the *Comparable* interface and the *equals()* method in each ADT. *(10 points)*
4. Use *HashMaps* and arrays to store, retrieve, and display information related to storms and their positions as described below. *(15 points)*
5. Use the *sort()* and *binarySearch()* methods from the Collections class to search for information related to the description below. *(20 points)*

- Develop and use a proper design. *(15 points)*
- Use proper documentation and formatting. *(15 points)*

Description:

An import skill in software design is extending the work you have done in a previous project. For this project you will rework Project 2, using the Java *HashMap* class and exception handling.

A database system stores and manages multiple pieces of information. Sometimes this information is related in simple or complex ways. For this project, you will implement a simple database of storms and cities. This database will allow you to look-up a list of storms that have come within a user-specified number of miles from a US City.

Your software will read in two data files that are both comma-delimited (.csv files) and store their information into the database. One data file holds information about Hurricanes, Tropical Storms, and Tropical Depressions; the other holds data about cities. For the storm data file, the information includes the name of the storm, the track of the storm (recorded as a series of latitude and longitude coordinates), the wind speed and severity (category) of the storm, and the date and time associated with each recorded location. For the city data file, the information includes the zip code of the city, the name of the city, the state in which the city is located, the location of the city (its latitude and longitude), its time zone, and whether the city is on daylight saving time.

Once the files have been read in, your program will enter a loop in which it asks the user for the name of two more files. This should be done using the technique described in the section below on reading input from the keyboard. The first of these two additional files will contain a list of queries for which your program should display information. Each query will be a pair consisting of a zip code and a distance. For each query, your program will find the city corresponding to the given zip code, then use that city's latitude and longitude to find all storms that came within the distance (specified by the second component of the query) to the city. The results will then be written to the screen, using System.out.println(), and written out to a user-specified output file. This output file will be the second of the two additional files the

user specifies. Each storm that came within the specified search range to the location specified will be listed in the output of the program, according to how far away the storm was in ascending order, e.g., the closest storm will be listed first and then by when the storm occurred. Once the results of running the program have been written to the screen and the output file, the user will be asked if he or she wishes to continue with the program or exit. If the user asks to continue, your program will return to the point in the loop at which it asks the user for the name of two more files (the query and output files).

The names of the input files that contain the storm and city data will be supplied on the command line by the user. The format of the command to run the program will be:

```
java Project5 <name of storm file> <name of city data file>
```

where `Project5` is the name of your project, `<name of storm file>` is the name of the file containing the storm information, and `<name of city data file>` is the name of the file containing the city data.

Learning Objectives:

HashMaps, Sorting and Retrieving:

Sorting information can be useful to users because the output may be organized in a way that makes it easier to use. It can also be useful to software developers because it can improve the efficiency of their software. Consider finding the cities based on their zip codes. If the data structure holding the city data is unsorted, you need to do a linear search through it to find a city. However, if the data structure is sorted based on zip code, you can do a binary search instead. A binary search will, in most cases, take far fewer comparisons to find the desired entry than a linear search.

Maps are useful for storing data as key-value pairs. For example, the key may be an integer representing the zip code and the value may be the city/zip object. In addition to providing a convenient data abstraction, maps can be implemented using a hashing function for fast lookup. Java provides the *HashMap* class which provides this functionality. It is important to note that the *HashMap* class is not particularly useful for sorting a list of objects.

To observe this efficiency gain, write three versions of the project. In the first version, leave the data structure holding the city data unsorted and search through it linearly when the user provides the query and output file names. In the second, sort this data structure based on zip code, then do binary searches on it. In the third, you must first add the key/value pairs for each city/zip to a *HashMap* and then retrieve the value by key. You will measure the amount of time the system uses in each case for internally handling city data. Internally handling city data for binary search includes adding the data to the data structure (list or *HashMap*), sorting and searching of it. ***Do not count the time the system uses for reading in the city data from the file or carrying out other activities, such as searching through the storm data or waiting for the user to provide input.*** Run each version of your program 5 times using only the sample provided query file one time for each run, record the values for times used for (1) inserting (2) sorting and (3) retrieving in each version, and present them in a simple table that includes totals and averages. An example of the table format is shown below.

Version 1	Insert Time	Sort Time	Retrieve Time	Total Time
Run 1				
Run 2				
Run 3				

```
    Run 4
    Run 5
    Average
Version 2           Insert Time       Sort Time   Retrieve Time          Total Time
    Run 1
    Run 2
    Run 3
    Run 4
    Run 5
    Average
Version 3           Insert Time       Sort Time   Retrieve Time          Total Time
    Run 1
    Run 2
    Run 3
    Run 4
    Run 5
    Average
```

If you only use the sample query file, does the first, second, or third version of your code spend less time internally handling city data? ***Why?*** If you use many additional query files, do you expect the first, second, or third version of your code to spend less time internally handling city data? ***Why?*** Rather than creating additional query files, try having your system repeatedly process the same query file. Is there some number of repetitions at which a less efficient version becomes a more efficient version? Is so, approximately what number is that?

Put the table of data and your answers to all of these questions into MILESTONES.txt under milestone 5.

File Formats:

Storm Input File:

An example set of data from the storm input file is shown below.

```
"YEAR","MONTH","DAY","TIME","ID","NAME","LATITUDE","LONGITUDE","WINDSPEED
  (KNOTS)","PRESSURE","CATEGORY","GENERAL LOCATION (BASIN)"
1949,6,11,"0000Z",1,"NOTNAMED",20.200,-106.300,45,0,"TS","East-Central Pacific"
1949,6,11,"0600Z",1,"NOTNAMED",20.200,-106.400,45,0,"TS","East-Central Pacific"
...
1952,7,20,"0600Z",25,"NOTNAMED",21.700,-112.900,45,0,"TS","East-Central Pacific"
1952,7,20,"1200Z",25,"NOTNAMED",22.000,-113.600,45,0,"TS","East-Central Pacific"
...
2002,9,19,"1800Z",1286,"ISIDORE",20.400,-81.700,65,983,"H1","North Atlantic"
```

```
2002,9,20,"0000Z",1286,"ISIDORE",20.700,-82.300,75,979,"H1","North Atlantic"
```

Fields:

- *Year* - The year the storm occurred.
- *Month* - The month the storm occurred.
- *Day* - The day of the month the storm occurred.
- *Time* - The time the location and associated information about the storm was captured. Note that each storm is listed as a sequence of measurements of the storm.
- *ID* - A unique integer ID associated with a storm, ranging from 1 to 1320. Note that ID numbers are not duplicated.
- *Name* - The name of the storm. Some storms do not have a name, in which case the name is recorded as NOTNAMED, and some storms have the same name as other storms.
- *Latitude* - The latitude the measurement was taken at.
- *Longitude* - The longitude the measurement was taken at.
- *Wind speed* - The wind speed at the time the measurement was taken at, recorded in Knots.
- *Pressure* - The pressure at the time the measurement was taken at. For some measurements a pressure measurement was not taken, in which case the pressure is recorded as zero.
- *Category* - The category of the storm: TD for tropical depression, TS for tropical storm, H1 for Hurricane Category 1, H2 for Hurricane Category 2, H3 for Hurricane Category 3, H4 for Hurricane Category 4, and H5 for Hurricane Category 5.
- *General Location* - The oceanic basin where the storm occurred, either North Atlantic or East-Central Pacific.

City Data File:

An example set of data from the city data file is given below:

```
"zip","city","state","latitude","longitude","timezone","dst"
"00210","Portsmouth","NH","43.005895","-71.013202","-5","1"
"00211","Portsmouth","NH","43.005895","-71.013202","-5","1"
...
```

Fields:
You should not have to worry about the fields that contain the timezone or daylight savings time (*dst*) information. Note that some cities have multiple zip codes. You must store all city and zip code combinations in case a search is made on an alternate zip code.

Output Format:

The text written to the screen and the output file for each storm matching the search criteria must follow the following output format.

```
Line 1: Zip code searched for
Line 2: Name of city and state
Line 3: An empty line
Line 4: Name of storm found
Line 5: Distance from zip code
Line 6: Basin the storm occurred in
Line 7 through n: Latitude, longitude of the storm along with the time, day, month,
and year the storm occurred, and the wind speed, pressure and severity (one line per
measurement).
Line n+1: Repeat Lines 3-n for the next storm found.
```

Storms should be listed according to how close they are to the zip code specified, in ascending order, and then by date in ascending order.

Sample Output (this is an example based on partially made up values):

```
Zip Code searched for: 70341
City and State: Belle Rose, LA

Andrew
10 Miles
North Atlantic
10.800 -35.500  0200  16-08-1992  25  1010  Tropical Depression
11.200 -37.400  1200  17-08-1992  30  1009  Tropical Depression
11.700 -39.600  1300  17-08-1992  30  1008  Tropical Depression
18.000 -56.900  1000  19-08-1992  45  1005  Tropical Storm
25.600 -67.000  1400  22-08-1992  65  994  Hurricane Category 1
25.700 -69.700  1600  22-08-1992  95  969  Hurricane Category 2
25.600 -71.100  1000  23-08-1992  110  961  Hurricane Category 3
25.500 -72.500  1300  23-08-1992  130  947  Hurricane Category 4
25.400 -74.200  1600  23-08-1992  145  933  Hurricane Category 5
29.200 -91.300  1000  26-08-1992  120  955  Hurricane Category 4
```

Design Issues:

Class size:

Beginning programmers tend to have a small number of classes with very large methods. The most extreme version of this is the programmer who puts everything in the main method of the program (sometimes to avoid using parameter passing). This is a disastrous approach. The main program becomes so large and unwieldy that it's hard to find and understand the code. Weird interactions between unrelated things can start happening, and pretty soon the program is impossible to understand and debug. This is an example of inadequate (or in the extreme case, missing) design.

Experienced programmers tend to have a large number of classes with small methods. The reason is that experienced programmers know that it's easier to debug smaller methods than large ones. An experienced programmer's main method will often be only 5-7 lines long. It will show the overall structure and organization of the application, and leave all of the details in the methods.

As you are programming, you need to make a conscious effort to work towards better structure and design. Focusing on making smaller methods and classes is usually a good step for most beginning programmers. Of course, the classes should be coherent and self contained, and not randomly organized. Additionally, interactions between classes should be minimized. Pairs of classes with lots of interactions are often a sign of poor class construction.

If you don't know how to pass parameters reliably and properly in Java, learn how to do it now. If you need help, come to office hours. It is impossible to program well if you can't pass parameters properly.

Implementation Issues:

File I/O:

To perform output to a file, use the *FileWriter* class with the *BufferedWriter* class as follows.

```
import java.io.*;
...
FileWriter     outfile = new FileWriter("filename");
BufferedWriter bw      = new BufferedWriter(outfile);
bw.write("Any string can go here");
bw.newLine();
```

When you have finished writing to a file, you must remember to close it, or the file won't be saved.

```
bw.close();
```

If you fail to close the file, it will be empty!

Instead of using add '*throws IOException*' as specified in Project 2, use a try/catch block to handle exceptional cases. For example, if the user specifies a file that does not exist, display an error message and request the filename again.

Reading Input from the Keyboard:
In order to get the location information from the user, you need to read input from the Keyboard. This can be done using the *InputStream* member of the *System* class, that is named *in*. When this input stream is wrapped with a *BufferedReader* object, the *readLine* method of the *BufferedReader* class can be used to read and store all of the characters typed by the user into a *String*. Note that *readLine* will block until the user presses the Enter key, i.e., the method call to *readLine* will not return until the user presses the Enter key.

The following code shows how to wrap and read strings from *System.in* using an *InputStreamReader* and a *BufferedReader*.

```
import java.io.*;
...
BufferedReader inputReader = new BufferedReader(
                                  new InputStreamReader( System.in ) );
...
System.out.print( "Type some input here: " );
String input = inputReader.readLine();  /* readline will not return until
                                         * the user presses Enter. */
System.out.println( "You typed: " +  input );
```

You need to add '*throws IOException*' to the signature of any method that uses or that directly or indirectly calls a method that uses a *BufferedReader* or *InputStreamReader*.

Calculating the Distance between two positions represented in Latitude and Longitude:
A short supplement will be posted on the class website along with this Project Requirements Handout on how to calculate the distance between two positions represented using latitude and longitude measurements.

Extra Credit Features:
You may extend this project for an extra 5 points of credit. Investigate the way in which the data is stored in the three versions of the city/zip search. For each version, display each list or *HashMap* to the screen (you don't need to turn in a hard copy) and answer the following questions. What order are the city/zip objects stored? Which version would be useful for displaying the sorted storm output? If you wanted to sort the entries of a *HashMap* how would you do it?

Finally, I suggest that you visit with the instructor if you have an idea for an extra credit feature in order to make sure it is acceptable.

Due Dates and Notes:

1. Your revised design and detailed Javadoc documentation are due on ***Wednesday, November 28th.*** Submit your revised UML design ***on engineering paper or using UML layout software*** at the ***beginning of class***. Submit the "stubbed" source code using the submit tool on *codd.cs.ou.edu* by ***9:00pm***. This submission counts as part of the design and documentation portions of the project grade. The commands

for submitting your "stubbed" Java source code on *codd.cs.ou.edu* are:

```
cd design2
/opt/cs2334/bin/submit cs2334-010 project5-design <.java filenames>
```

where `<.java filenames>` is a list of the `.java` files you are submitting.

2. The final version of the project is due on ***Wednesday, December 5th.*** Submit your final UML design ***on engineering paper or using UML layout software*** at the ***beginning of class***. Submit your source code files and the COMPILATION.txt, EXECUTION.txt, and MILESTONES.txt files using the submit tool on *codd.cs.ou.edu* by ***9:00pm.*** The commands for submitting your final project on *codd.cs.ou.edu* are:

```
cd project5
/opt/cs2334/bin/submit cs2334-010 project5-final <.java filenames>
```

where `<.java filenames>` is a list of the `.java` files you are submitting. Make sure that the COMPILATION.txt, EXECUTION.txt, and MILESTONES.txt files are present in the directory along with all of your `.java` source files.

3. The *Get Out of Jail Free* due date for this project is ***Friday, December 7th***. Submit your final UML design ***on engineering paper or using UML layout software*** at the ***beginning of class***. Submit your source code files and the COMPILATION.txt, EXECUTION.txt, and MILESTONES.txt files using the submit tool on *codd.cs.ou.edu* by ***9:00pm***. The commands for submitting your final project on *codd.cs.ou.edu* using Get Out of Jail Free are:

```
cd project5
/opt/cs2334/bin/submit cs2334-010 project5-goojf <.java file names>
```

where `<.java filenames>` is a list of the `.java` files you are submitting. Make sure that the COMPILATION.txt, EXECUTION.txt, and MILESTONES.txt files are present in the directory along with all of your .java source files.

4. You are not allowed to use the StringTokenizer class. Instead ***you must use String.split()*** and a regular expression that specifies the delimiters you wish to use to "tokenize" or split each line of the file.

5. You may write your program from scratch or may start from programs for which the source code is freely available on the web or through other sources (such as friends or student organizations). If you do not start from scratch, you ***must*** give a complete and accurate accounting of where all of your code came from and indicate which parts are original or changed, and which you got from which other source. Failure to give credit where credit is due is academic fraud and will be dealt with accordingly.

6. As noted in the syllabus, you are required to work on this programming assignment in a group of at least two people. It is your responsibility to find other group members and work with them. The group should turn in only one (1) hard copy and one (1) electronic copy of the assignment. Both the electronic and hard copies should contain the names and student ID numbers of ***all*** group members. If your group composition changes during the course of working on this assignment (for example, a group of five splits into a group of two and a separate group of three), this must be clearly indicated in your write-up, including the names and student ID numbers of everyone involved and details of when the change occurred and who accomplished what before and after the change.

Each group member is required to contribute equally to each project, as far as is possible. You must thoroughly document which group members were involved in each part of the project. For example, if you have three functions in your program and one function was written by group member one, the second was written by group member two, and the third was written jointly and equally by group members three and four, both your write-up and the comments in your code must clearly indicate this division of labor.